Hardware Shading with ATI_vertex_shader and ATI_fragment_shader

Evan Hart
Ehart@ati.com

Jason L. Mitchell
JasonM@ati.com

3D Application Research Group
ATI Research

Introduction

Programmable shaders are a powerful way to describe the interaction of surfaces with light, as evidenced by the success of programmable shading models like RenderMan and others. As graphics hardware evolves beyond the traditional "fixed function" pipeline, hardware designers are looking to programmable models to empower the next generation of real-time content. To allow content to interface with programmable hardware, we have designed shader extensions to OpenGL which operate at the vertex and fragment levels. In these notes, we will outline the behavior of the **`ATI_vertex_shader`** and **`ATI_fragment_shader`** extensions as examples of programmable interfaces designed for real-time graphics. While we will use the syntax of these two extensions throughout these notes, we will discuss several issues of general interest to anyone who is specifying, implementing or using a programmable 3D graphics API in a production environment.

While we expect many programmers to write to these extensions directly in C, we also expect to see a set of metaprogramming tools made available to drive the models. The `ATI_vertex_shader` API, for example, turns out to look very much like the intermediate representation in the Stanford Real-time Shading Language [Proudfoot01]. By providing some support for subroutines, `ATI_vertex_shader` also allows an application to separate out *light*, *surface* and *atmospheric* shaders, which has proven to be useful in other models like Cook's Shade Trees [Cook84] and Pixar's RenderMan [Upstill88] [Hanrahan90]. Although `ATI_vertex_shader` and `ATI_fragment_shader` are currently separate extensions, they are designed to have similar interfaces and to eventually be merged into one model.

Goals of the **`ATI_vertex_shader`** and **`ATI_fragment_shader`** extensions:

- Enable programmability in the graphics pipeline
- Easily migrate to future hardware designs
- Work together in a clean and intuitive way
- Have enough generality to be implemented on a wide range of hardware, thus encouraging multi-vendor interest and support
- Address complexity issues by providing some subroutine and looping capabilities
- Free apps from the responsibility for managing constant and variable storage space
- Prevent app from having to re-implement a part of the pipe that it isn't modifying
    - Shader library provides pluggable functionality for lighting, texture coordinate generation, etc.

Vertex operations with ATI_vertex_shader

Like other programmable 3D graphics APIs, `ATI_vertex_shader` allows substitution of a flexible per-vertex programming model into the graphics pipeline in place of the traditional fixed-function pipeline (**Figure 1**). This can be toggled so that primitives which do not require the additional flexibility of the programmable pipeline can use the existing fixed functionality.

As shown in **Figure 1**, vertex shading happens after higher-order surface tessellation. This allows the shading operations to operate on the "high frequency" vertex data coming out of a tessellation stage such as a Curved PN Triangle implementation [Vlachos01]. In the case of vertex Phong lighting computations, this results in more well-defined specular highlights.

The vertex shading stage feeds into the clipping stage, which applies frustum and user clip planes in clip space. These clipped triangles are then passed to the triangle setup stage, followed by rasterization using `ATI_fragment_shader`.

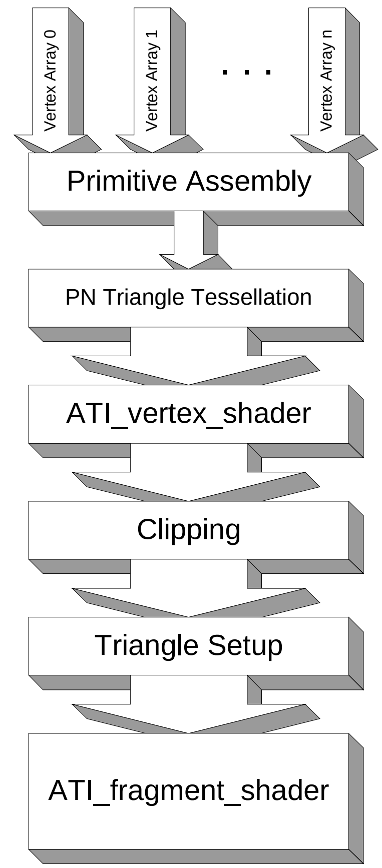

Figure 1 `ATI_vertex_shader` and `ATI_fragment_shader` in the graphics pipeline

Creating a shader with ATI_vertex_shader

In `ATI_vertex_shader`, creation of a vertex shader is done in the usual OpenGL manner, with the implementation generating a number of shader names that an application can bind for use or delete when no longer required:

```
uint glGenVertexShadersATI (GLuint range)
void glBindVertexShaderATI (GLuint id)
void glDeleteVertexShaderATI(GLuint id)
```

Specification of a vertex shader is bracketed by calls to `glBeginVertexShaderATI()` and `glEndVertexShaderATI()`, much like a display list:

```
glBindVertexShaderATI(simpleVertexShader);
glBeginVertexShaderATI();
// declare variables, instructions etc
glEndVertexShaderATI();
```

When the application wishes to use a given vertex shader in place of the fixed function transformation pipeline, the shader is bound, and programmable shading is enabled as follows:

```
glBindVertexShaderATI(simpleVertexShader);
glEnable(GL_VERTEX_SHADER_ATI);
```

To switch back to the fixed function transformation pipeline, the application disables vertex shading:

```
glDisable(GL_VERTEX_SHADER_ATI);
```

Simple Vertex Shader

Before describing the instruction set, storage types and other aspects of the programming model, we show a simple shader to give a sense of its structure:

```
//Initialize global parameter bindings
Modelview  = glGenDataATI (GL_MATRIX_ATI, GL_INVARIANT_ATI, GL_FULL_RANGE_ATI, 1);
Projection = glGenDataATI (GL_MATRIX_ATI, GL_INVARIANT_ATI, GL_FULL_RANGE_ATI, 1);
Vertex     = glGenDataATI (GL_VECTOR_ATI, GL_VARIANT_ATI,   GL_FULL_RANGE_ATI, 1);
Normal     = glGenDataATI (GL_VECTOR_ATI, GL_VARIANT_ATI,   GL_FULL_RANGE_ATI, 1);

glBindParameterATI (GL_MODELVIEW_MATRIX, Modelview);
glBindParameterATI (GL_PROJECTION_MATRIX, Projection);
glBindParameterATI (GL_CURRENT_VERTEX_ATI, Vertex);
glBindParameterATI (GL_CURRENT_NORMAL_ATI, Normal);

glBindVertexShaderATI (xform); //a simple diffuse shader
glBeginVertexShaderATI();
{
   float  direction[4] = { 0.57735f, 0.57735f, 0.57735f, 0.0f}; //direction vector (1,1,1) normalized
   float  material[4]  = { 1.00000f, 1.00000f, 0.00000f, 1.0f}; //yellow diffuse material
   float  ambient[4]   = { 0.20000f, 0.20000f, 0.20000f, 0.0f}; //scene ambient light intensity
   GLuint lightDirection;
   GLuint diffMaterial;
   GLuint sceneAmbient;
   GLuint eyeVertex;
   GLuint clipVertex;
   GLuint eyeNormal;
   GLuint intensity;

   // generate local values
   eyeVertex  = glGenDataATI (GL_VECTOR_ATI, GL_LOCAL_ATI, GL_FULL_RANGE_ATI, 1);
   clipVertex = glGenDataATI (GL_VECTOR_ATI, GL_LOCAL_ATI, GL_FULL_RANGE_ATI, 1);
   eyeNormal  = glGenDataATI (GL_VECTOR_ATI, GL_LOCAL_ATI, GL_FULL_RANGE_ATI, 1);
   intensity  = glGenDataATI (GL_VECTOR_ATI, GL_LOCAL_ATI, GL_FULL_RANGE_ATI, 1);

   // generate constant values
   lightDirection = glGenDataATI (GL_VECTOR_ATI, GL_LOCAL_CONSTANT_ATI, GL_FULL_RANGE_ATI, 1);
   diffMaterial   = glGenDataATI (GL_VECTOR_ATI, GL_LOCAL_CONSTANT_ATI, GL_FULL_RANGE_ATI, 1);
   sceneAmbient   = glGenDataATI (GL_VECTOR_ATI, GL_LOCAL_CONSTANT_ATI, GL_FULL_RANGE_ATI, 1);

   glSetShaderStateATI (lightDirection, GL_FLOAT, direction);
   glSetShaderStateATI (diffMaterial,   GL_FLOAT, material);
   glSetShaderStateATI (sceneAmbient,   GL_FLOAT, ambient);

   glShaderOp2ATI (GL_OP_MULTIPLY_MATRIX_ATI, eyeVertex, Modelview, Vertex);
   glShaderOp2ATI (GL_OP_MULTIPLY_MATRIX_ATI, clipVertex, Projection, eyeVertex);
   glShaderOp1ATI (GL_OP_EXPORT_ATI, GL_OUTPUT_VERTEX_ATI, clipVertex);

   // assumes no scaling/shearing in modelview matrix
   glShaderOp2ATI (GL_OP_MULTIPLY_MATRIX_ATI, eyeNormal, Modelview, Normal);
   glShaderOp2ATI (GL_OP_DOT3_ATI, intensity, lightDirection, eyeNormal);
   glShaderOp2ATI (GL_OP_ADD_ATI, intensity, sceneAmbient, intensity);
   glShaderOp2ATI (GL_OP_MUL_ATI, intensity, diffMaterial, intensity);

   glShaderOp1ATI (GL_OP_EXPORT_ATI, GL_OUTPUT_COLOR0_ATI, intensity);
}
glEndVertexShaderATI();
```

Vertex Shader Structure

The only operations allowed between `glBeginVertexShaderATI()` and `glEndVertexShaderATI()` are `glGenDataATI()`, `glSetShaderStateATI()`, `glShaderOpxATI()`, `glSwizzleATI()` and `glWriteMaskATI()`.

As you can probably infer from the sample code above, `ATI_vertex_shader` is a 4D vector programming language. We will now describe the use of local constants and variables in the language.

Local Constants and Variables

Like any high level language, a shader written using `ATI_vertex_shader` must declare its constants and variables. This is done with the `glGenDataATI()` entrypoint:

```
uint glGenDataATI(enum datatype, enum storagetype, enum range, uint components)
```

Each constant or variable can be of data type `GL_SCALAR_ATI`, (4D) `GL_VECTOR_ATI` or (4x4) `GL_MATRIX_ATI` and can have a storage type of `GL_VARIANT_ATI`, `GL_INVARIANT_ATI`, `GL_LOCAL_CONSTANT_ATI` or `GL_LOCAL_ATI`. We will discuss variants later—for now, we will focus on local variables and constants. The simple diffuse lighting shader above creates 4 local vector variables and 3 local vector constants. The four variables are not initialized, while the three constants are initialized using the `glSetShaderStateATI()` entrypoint:

```
glSetShaderStateATI (lightDirection, GL_FLOAT, direction);
glSetShaderStateATI (diffMaterial,   GL_FLOAT, material);
glSetShaderStateATI (sceneAmbient,   GL_FLOAT, ambient);
```

After initialization, the shader is free to read from any of these constants, though it may not write to them. Reading from either of the declared variables without first writing to it will result in failed creation of the shader and `glEndVertexShader()`will return an error.

In addition to the ability to declare generic constants and variables, a vertex shader can access useful OpenGL state by binding variables or constants to OpenGL states. This is convenient for applications that mix use of the fixed-function vertex pipeline with use of `ATI_vertex_shader` or just as a simple means for managing common quantities such as a modelview matrix.

Accessing OpenGL State

Once a variable or constant has been declared, it may be bound to some specific OpenGL state for convenience. Relevant OpenGL state may be bound to a constant as shown in the `ATI_vertex_shader` spec. There are special entrypoints for binding state related to lighting, materials, texgen and texture contexts since these are contextual states. OpenGL state that is contextual is accessed through these four special context-sensitive parameter binding entrypoints in order to utilize the existing enums and prevent an enum explosion.

```
void glBindParameterATI (enum value, uint name)
void glBindLightParameterATI (enum light, enum value, uint name)
void glBindMaterialParameterATI (enum face, enum value, uint name)
void glBindTexGenParameterATI (enum coord, enum value, uint name)
void glBindTextureParameterATI (enum coord, enum value, uint name)
```

Instruction set

There are 25 opcodes available in `ATI_vertex_shader`, as shown in **Table 1** below:

OP_EXPORT_ATI	OP_INDEX_ATI	OP_NEGATE_ATI
OP_MOV_ATI	OP_MULTIPLY_MATRIX_ATI	OP_DOT3_ATI
OP_DOT4_ATI	OP_MUL_ATI	OP_ADD_ATI
OP_MADD_ATI	OP_FRAC_ATI	OP_MAX_ATI
OP_MIN_ATI	OP_SET_GE_ATI	OP_SET_LT_ATI
OP_CLAMP_ATI	OP_FLOOR_ATI	OP_ROUND_ATI
OP_EXP_BASE_2_ATI	OP_LOG_BASE_2_ATI	OP_POWER_ATI
OP_RECIP_ATI	OP_RECIP_SQRT_ATI	OP_SUB_ATI
OP_CROSS_PRODUCT_ATI		

Table 1 - `ATI_vertex_shader` opcodes

As shown in the sample shader above, different operations have different numbers of arguments. This is handled in the usual OpenGL manner with the `glShaderOp`*x*`ATI() entrypoints`:

```
glShaderOp1ATI (enum op, uint res, uint arg1)
glShaderOp2ATI (enum op, uint res, uint arg1, uint arg2)
glShaderOp3ATI (enum op, uint res, uint arg1, uint arg2, uint arg3)
```

Micro Operations versus Instructions

As in any programmable processor, a single instruction may translate into some number of micro operations, clock cycles etc. Micro op counts are implementation details that we expect to vary between vendors and generations of hardware, much like they do for CPUs.

Argument Swizzling and Modification

Components of arguments may be individually swizzled and/or negated. Literal 1's and 0's may also be substituted for any component. This is expressed with the glSwizzleATI() entrypoint. For example, to replicate the X component of an argument across all 4 components, one would use the following syntax:

```
glSwizzleATI (res, in, X_ATI, X_ATI, X_ATI, X_ATI);
```

To negate only the X component, one would use:

```
glSwizzleATI (res, in, NEGATIVE_X_ATI, Y_ATI, Z_ATI, W_ATI);
```

All 11 argument modifiers are shown in **Table 2** below. Any combination of these arguments may be passed to `glSwizzleATI()`.

X_ATI	Y_ATI	Z_ATI	W_ATI
NEGATIVE_X_ATI	NEGATIVE_Y_ATI	NEGATIVE_Z_ATI	NEGATIVE_W_ATI
ZERO_ATI	ONE_ATI	NEGATIVE_ONE_ATI	

Table 2 – Argument modifiers for `ATI_vertex_shader`

Write masks

It is possible to mask writes to destination registers using the `glWriteMaskATI()` entrypoint. Only `GL_TRUE` or `GL_FALSE` can be passed as the last four parameters to `glWriteMaskATI()`. For example, to write only to the X component of the destination, use the following syntax:

```
glWriteMaskATI (res, in, GL_TRUE, GL_FALSE, GL_FALSE, GL_FALSE);
```

Swizzles and Write Masks as Free Instructions

Although the swizzle and write mask operations are expressed as discrete operations, we expect implementations of `ATI_vertex_shader` to collapse them into what amount to modifiers to the operations performed by a call to `glShaderOpxATI()`. For example, we expect the following kind of call sequence to collapse into one hardware instruction, making the swizzles and write masks "free":

```
glSwizzleATI (swarg1  , arg1, ... );
glSwizzleATI (swarg1  , arg1, ...  );
glSwizzleATI (swarg1  , arg1, ...   );
glShaderOp3ATI (MAD, winput, sarg1, sarg2, sarg3);
glWriteMaskATI ( output, winput );
```

After exploring ways to express these swizzles and write masks in the `glShaderOpxATI()` entrypoints, we determined that doing so would result in absurdly verbose entrypoints as well as enumerant explosion. As a result, we settled on the approach described above. This is manageable for the C programmer and will surely be expressed very concisely in the metaprogramming tools that will layer on top of `ATI_vertex_shader`.

Variants: Custom Vertex Attributes

The simple shader above uses traditional OpenGL inputs with a defined semantic meaning; specifically vertex position and normal. Here, we will explain how we allow the application to specify custom vertex attributes via immediate mode and vertex array interfaces. Earlier, we showed how `glGenDataATI()` can be used to declare local constants and variables. The same entrypoint is also used to declare custom vertex attributes, which we call *variants*. To declare a single full-range vector variant:

```
var1 = glGenDataATI (GL_VECTOR_ATI, GL_VARIANT_ATI, GL_FULL_RANGE_ATI, 1);
```

If rendering using immediate mode, the application can specify the values of this variant for each vertex using `glSetVariantATI()`. If rendering with vertex arrays, the application can specify the values of the variant in the following manner:

```
glSetVariantPointerATI (var1, stride, type, addr)
glEnableVariantClientStateATI (var1);
// render
glDisableVariantClientStateATI (var1)
```

The variant `var1` can be read in the vertex shader, but may not be written. It can be thought of as another interpolated quantity like position or normal, but with no inherent semantic meaning.

Data Types, Normalized Range and Full Range

When specifying data to OpenGL, the data may be defined to be a tuple of any of the following types: `GL_DOUBLE`, `GL_FLOAT`, `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT` or GL_UNSIGNED_INT. Clearly, a variant may also take any of these types. For variant types other than GL_DOUBLE or GL_FLOAT, it is convenient to have the flexibility to specify whether the data is intended to represent a range from -1.0 to 1.0 or the native range of the data type (i.e. -128 to 127 for GL_BYTE). For example, an application using the constant store as a palette of matrices for character animation might use a number of full-range ubyte variants to index into the palette. In another situation, an application may realize storage and bandwidth savings by quantizing a normalized vector component of a vertex. In this case, the application would use a normalized-range byte variant. When reading from the normalized variant in the vertex shader, the data can be assumed to be in the range of -1.0 to 1.0. This is analogous to the way that OpenGL currently handles the differences between colors, texture coordinates, vertices and normals.

Vertex Shader Outputs

The contents of special output registers constitute the output of the vertex shader. The ATI_vertex_shader extension defines a set of output registers that may be written into by the vertex shader, but may not be read. These include the clip-space position of the vertex (4D), two 4D colors, a scalar fog factor and some number of 4D texture coordinates. (Point size for point sprites, and any other special vertex shader outputs are specified in their own separate extensions.) Any output registers that are not written to by the vertex shader code are undefined and should not be consumed in the fragment shader. As an example, an implementation that supports six textures could consume the contents of the following 10 output registers in the fragment shader:

OUTPUT_VERTEX_ATI	OUTPUT_FOG_ATI
OUTPUT_COLOR0_ATI	OUTPUT_COLOR1_ATI
OUTPUT_TEXTURE_COORD0_ATI	OUTPUT_TEXTURE_COORD1_ATI
OUTPUT_TEXTURE_COORD2_ATI	OUTPUT_TEXTURE_COORD3_ATI
OUTPUT_TEXTURE_COORD4_ATI	OUTPUT_TEXTURE_COORD5_ATI

Table 3 – Vertex Shader Output Registers

For example, the simple shader illustrated earlier transforms the input position into a local variable and then exports that local variable with the GL_OP_EXPORT_ATI operation:

```
glShaderOp1ATI (GL_OP_EXPORT_ATI, GL_OUTPUT_VERTEX_ATI, clipVertex);
```

After performing lighting calculations, the shader exports another local variable to color0 of the vertex:

```
glShaderOp1ATI (GL_OP_EXPORT_ATI, GL_OUTPUT_COLOR0_ATI, intensity);
```

Clip Planes

As shown in Figure 1, the output of the vertex shader is in clip-space. This means that frustum and user clip planes can be applied to primitives whose vertices are in this space since eye-space is now undefined. The algorithm is the same, except the half-space is now defined as:

$$[p'_1\ p'_2\ p'_3\ p'_4]\quad P_{inv}\quad \begin{bmatrix} \boldsymbol{x}_{clip} \\ \boldsymbol{y}_{clip} \\ \boldsymbol{z}_{clip} \\ \boldsymbol{w}_{clip} \end{bmatrix} \geq 0$$

Where P is the projection matrix and $\boldsymbol{x}_{clip}$, $\boldsymbol{y}_{clip}$, $\boldsymbol{z}_{clip}$, and $\boldsymbol{w}_{clip}$ are the clip-space vertex coordinates. When P is singular, the result of clipping is undefined.

Case Study: Using a subroutine for lighting

The functionality illustrated in the sample shader above can be simplified further by the use of a subroutine to compartmentalize the lighting calculations. The code below illustrates `ATI_vertex_shader`'s ability to support inline subroutines and unrolled loops.

```
//
// Function to compute diffuse illumination for a point light
//
//    This takes the eye-space normal, vertex, and light position.
// It computes the intensity of the light, modulates by the light color,
// and adds to the accumulated intensity
//
////////////////////////////////////////////////////////////////////////
void PointDiffuse( GLuint light, GLuint normal, GLuint vertex, GLuint color,
                   GLuint intensity)
{

   GLuint lightDirection;
   GLuint lightIntensity

   //generate local values
   lightDirection = glGenDataATI( GL_VECTOR_ATI, GL_LOCAL_ATI,
                                  GL_FULL_RANGE_ATI, 1);
   LightIntensity = glGenDataATI( GL_SCALAR_ATI, GL_LOCAL_ATI,
                                  GL_FULL_RANGE_ATI, 1);

   glTransformOp2ATI( GL_OP_SUBTRACT_ATI, lightDirection, light, vertex);
   glTransformOp2ATI( GL_OP_DOT3_ATI, lightIntensity, light, normal );

   glTransformOp3ATI( GL_OP_MADD_ATI, intensity, lightIntensity, color,
                      intensity);

}

void DefineSimpleDiffuseShader( int numLights, GLuint *lightPos,
                                GLuint *lightColor, . . . )
{
   glBeginVertexShader();
   //Setup local variables
   . . .

   //Transform components to eye-space
   . . .

   //Set ambient light value
   . . .

   for ( int i = 0; i < numLights; i++)
   {
      PointDiffuse( lightPos[i], eyeNormal, eyeVertex, lightColor[i],
                    SigmaLight );
   }

   //Modulate light with material
   . . .

   //Output components
   . . .
   glEndVertexShader();
}
```

Fragment operations with ATI_fragment_shader

Like its counterpart **ATI_vertex_shader**, the **ATI_fragment_shader** extension provides a means for inserting a flexible per-pixel programming model into the graphics pipeline in place of the traditional multitexture pipeline (**Figure 1**). This can be toggled so that primitives which do not require the additional flexibility of the programmable pipeline can use the existing fixed functionality provided by extensions such as ARB_multitexture, ARB_texture_env_combine, EXT_texture_env_dot3 and others.

The ATI_fragment_shader extension provides a very general means of expressing fragment color blending and dependent texture address modification. The programming model is a register-based model and the C syntax is similar to the ATI_vertex_shader extension described above. The number of instructions, texture lookups, read/write registers and constants is queryable, to provide easy migration between hardware generations.

One advantageous property of the model is a unified instruction set used throughout the shader. That is, the same instructions are provided when operating on address or color data. In fact, the distinction between address and color data becomes somewhat meaningless in this setting. Additionally, this unified approach gives application programmers a single instruction set to learn and eliminates the awkward CISC address shading "modes" found in other models. It really is possible to "just do some math ops and look the result up in a texture."

Creating a shader with ATI_fragment_shader

As in ATI_vertex_shader above, creation of a fragment shader is done in the usual OpenGL manner, with the implementation generating a number of shader names that an application can bind for use or delete when no longer required:

```
uint glGenFragmentShadersATI (GLuint range)
void glBindFragmentShaderATI (GLuint id)
void glDeleteFragmentShaderATI(GLuint id)
```

As above, specification of a fragment shader is bracketed by calls to glBeginFragmentShaderATI() and glEndFragmentShaderATI():

```
glBindFragmentShaderATI(simpleFragmentShader);
glBeginFragmentShaderATI();
// declare variables, instructions etc
glEndFragmentShaderATI();
```

When the application wishes to use a given fragment shader in place of the multitexture pipeline, the shader is bound and programmable shading is enabled as follows:

```
glBindFragmentShaderATI(simpleFragmentShader);
glEnable(GL_FRAGMENT_SHADER_ATI);
```

To switch back to the multitexture pipeline, the application disables fragment shaders:

```
glDisable(GL_FRAGMENT_SHADER_ATI);
```

Sample Fragment Shader

```
// Base/Bump 2D coords in tex coord 0
// Tangent space half angle in 3D tex coord 2
// Tangent space light vector in 3D tex coord 5
// Light's Diffuse color set in constant 0 and specular color in constant 1

simpleFragmentShader = glGenFragmentShadersATI(1);
glBindFragmentShaderATI (simpleFragmentShader);
glBeginFragmentShaderATI();
    glSampleMapATI ( 0, GL_INTERP_0_ATI);    // Sample the N map (k in alpha)
    glSampleMapATI ( 2, GL_INTERP_2_ATI);    // Normalize H with a cube map

    // N.H
    glColorFragmentOp2ATI (GL_DOT3_ATI, GL_REG_2_ATI, GL_RED_BIT_ATI, GL_NONE,
                                        GL_REG_0_ATI, GL_NONE, GL_BIAS_BIT_ATI|GL_2X_BIT_ATI,
                                        GL_REG_2_ATI, GL_NONE, GL_BIAS_BIT_ATI|GL_2X_BIT_ATI);

    // grab the k channel
    glColorFragmentOp1ATI (GL_MOV_ATI, GL_REG_2_ATI, GL_GREEN_BIT_ATI, GL_NONE,
                                       GL_REG_0_ATI, GL_ALPHA, GL_NONE);

    // End of first pass
    // Dependent texture reads to raise (N.H) to k power
    glSampleMapATI ( 1, GL_INTERP_0_ATI);             // base
    glSampleMapATI ( 3, GL_REG_2_ATI);                // (N.H)^k
    glSampleMapATI ( 2, GL_INTERP_1_ATI);             // Normalize L

    // N.L
    glColorFragmentOp2ATI (GL_DOT3_ATI, GL_REG_0_ATI, GL_NONE, GL_SATURATE_BIT_ATI,
                                        GL_REG_0_ATI, GL_NONE, GL_BIAS_BIT_ATI|GL_2X_BIT_ATI,
                                        GL_REG_2_ATI, GL_NONE, GL_BIAS_BIT_ATI|GL_2X_BIT_ATI);

    // ((N.H)^k) * gloss)
    glColorFragmentOp2ATI (GL_MUL_ATI, GL_REG_2_ATI, GL_NONE,  GL_NONE,
                                       GL_REG_3_ATI, GL_NONE,  GL_NONE,
                                       GL_REG_1_ATI, GL_ALPHA, GL_NONE);

    // (N.L) * diffuse_color
    glColorFragmentOp2ATI (GL_MUL_ATI, GL_REG_0_ATI, GL_NONE, GL_NONE,
                                       GL_REG_0_ATI, GL_NONE, GL_NONE,
                                       GL_CON_0_ATI, GL_NONE, GL_NONE);

    // ((N.H)^k) * gloss) * specular_color
    glColorFragmentOp2ATI (GL_MUL_ATI, GL_REG_2_ATI, GL_NONE, GL_NONE,
                                       GL_REG_2_ATI, GL_NONE, GL_NONE,
                                       GL_CON_1_ATI, GL_NONE, GL_NONE);

    // Result = (N.L) * diffuse_color * base + ((N.H)^k) * gloss * specular_color)
    glColorFragmentOp3ATI (GL_MAD_ATI, GL_REG_0_ATI, GL_NONE, GL_SATURATE_BIT_ATI,
                                       GL_REG_0_ATI, GL_NONE, GL_NONE,
                                       GL_REG_1_ATI, GL_NONE, GL_NONE,
                                       GL_REG_2_ATI, GL_NONE, GL_NONE);
glEndFragmentShaderATI();
```

Fragment Shader Structure

The only operations allowed between `glBeginFragmentShaderATI()` and `glEndFragmentShaderATI()` are `glPassTexCoordATI()`, `glSampleMapATI()`, `glColorFragmentOp`*x*`ATI()`, `glAlphaFragmentOp`*x*`ATI()` and `glSetFragmentShaderConstantATI()`. The `glSetFragmentShaderConstantATI()` entrypoint may also be used outside of the fragment shader, to effectively specify constant parameters to the shader. This is what is expected to be done in the sample shader above, as the shader assumes that the diffuse and specular colors of the given light source are stored in constants zero and one when the shader executes.

Registers, Constants and Interpolators

Although the creation and management of fragment shaders is the same as vertex shaders, we have chosen not to abstract the constant and register management like we did in `ATI_vertex_shader`. This is because we expect hardware implementations to have far fewer registers and constants available at the fragment level than at the vertex level, necessitating more of a "hand-tuned" approach by content developers. An implementation will support some number of read-only constants and read-write registers. The number of constants and registers supported is queryable by passing `NUM_FRAGMENT_REGISTERS_ATI` or `NUM_FRAGMENT_CONSTANTS_ATI` to `glGet()`. The primary and secondary colors are also available as read-only interpolated data.

Constants may be declared within a shader by making up to `NUM_FRAGMENT_CONSTANTS_ATI` calls to `glSetFragmentShaderConstantATI()` immediately after `glBeginFragmentShader()`. In this case, the constants override the constants that are part of the global OpenGL state during the time that the shader is bound. Constants may be read at any point in the fragment shader, even if they are not declared within the shader. When not declared within the shader, the constants can be thought of as parameters to the shader. In the sample shader above, the application is expected to set constant zero and one according to the properties of the light causing the surface bumps.

Texture Sampling and Coordinate Routing

Prior to performing arithmetic instructions in the fragment shader, maps may be sampled and texture coordinates may be routed into registers. These operations are accomplished with the `glPassTexCoordATI()` and `glSampleMapATI()` operations.

Instruction Set

`ATI_fragment_shader` provides for multiple shading passes separated by texture sampling. (This is not to be confused with multi-pass rendering, as the frame buffer is not updated between passes.) The total number of shader passes provided by an implementation is a queryable value accessible through glGet. Additionally, the maximum number of operations allowed during a pass is also queryable. The following entrypoints are used to specify arithmetic instructions within the pixel shader.

```
void glColorFragmentOpxATI (enum op, uint dst,  uint dst_mask,  uint dst_scale
                                     uint arg1, uint arg1_repl, uint arg1_mod
                                     . . . )


void glAlphaFragmentOpxATI (enum op, uint dst,  uint dst_mask,  uint dst_scale
                                     uint arg1, uint arg1_repl, uint arg1_mod
                                     . . .)
```

op can be one of `GL_ADD_ATI`, `GL_SUB_ATI`, `GL_MUL_ATI`, `GL_MAD_ATI`, `GL_LERP_ATI`, `GL_MOV_ATI`, `GL_CND_ATI`, `GL_CND0_ATI`, `GL_DOT3_ATI`, `GL_DOT4_ATI`, `GL_FRAC_ATI` or `GL_DOT2_ADD_ATI`.

dst can be one of `GL_REG_0_ATI`, `GL_REG_1_ATI`, `GL_REG_2_ATI . . . GL_REG_`*`N-1`*`_ATI` where *N* is `GL_NUM_FRAGMENT_REGISTERS`

dst_mask can be any bitwise or'ing of GL_RED_BIT_ATI, GL_GREEN_BIT_ATI, GL_BLUE_BIT_ATI or GL_ALPHA_BIT_ATI. For no masking, use GL_NONE_ATI.

dst_scale can be GL_SATURATE_ATI optionally bitwise or'd with one of GL_2X_BIT_ATI, GL_4X_BIT_ATI, GL_8X_BIT_ATI, GL_HALF_BIT_ATI, GL_QUARTER_BIT_ATI or GL_EIGHTH_BIT_ATI. For no saturation or scaling, use GL_NONE_ATI.

arg*n* can be one of GL_REG_0_ATI, GL_REG_1_ATI, GL_REG_2_ATI . . . GL_REG_*N-1*_ATI, GL_CON_0_ATI, GL_CON_1_ATI, GL_CON_2_ATI . . . GL_CON_*M-1*_ATI, GL_PRIMARYCOLOR or GL_SECONARYCOLOR_ATI.

where *N* is GL_NUM_FRAGMENT_REGISTERS_ATI
where *M* is GL_NUM_FRAGMENT_CONSTANTS_ATI

arg*n*_repl can be one of GL_RED_ATI, GL_GREEN_ATI, GL_BLUE_ATI, GL_ALPHA_ATI or GL_NONE_ATI

arg*n*_mod can be GL_COMP or any bitwise or'ing of GL_NEGATE_BIT_ATI, GL_BIAS_BIT_ATI or GL_2X_BIT_ATI. For no source modifications, use GL_NONE.

coord can be any integer from 0 to GL_MAX_TEXTURES-1

map can be any integer from 0 to GL_MAX_TEXTURES-1

Argument Replication and Modification

As noted above, there are four mutually exclusive options for channel replication of arguments. Replication is independent of other argument modification.

Alpha Replicate	GL_ALPHA	Replicates the alpha channel to all colors
Red Replicate	GL_RED	Replicates the red channel to all colors
Green Replicate	GL_GREEN	Replicates the green channel to all colors
Blue Replicate	GL_BLUE	Replicates the blue channel to all colors

As noted above, there are four additional options for argument modification. This these may happen independent of any replication of the data.

Complement	GL_COMP_BIT_ATI	Complements y = 1.0 - x
Negate	GL_NEGATE_BIT_ATI	Negates the value y = -x
Bias	GL_BIAS_BIT_ATI	Shifts value down by ½, y = (x-0.5)
Scale x 2	GL_2X_BIT_ATI	Scales input by 2

Note the following rules for combining modifiers:

- GL_RED, GL_GREEN, GL_BLUE and GL_ALPHA are mutually exclusive
- GL_COMP is exclusive of GL_BIAS, GL_NEGATE and GL_2X

When using multiple argument modifiers, GL_BIAS_ATI happens first, followed by GL_2X_ATI and GL_NEGATE_ATI.

Write Modifiers

The following destination modifiers are available to modify the results of the calculation before writing them into the destination register:

GL_2X_BIT_ATI	Multiply result by 2
GL_4X_BIT_ATI	Multiply result by 4
GL_8X_BIT_ATI	Multiply result by 8
GL_HALF_ATI	Divide result by 2
GL_QUARTER_ATI	Divide result by 4
GL_EIGHTH_ATI	Divide result by 8

GL_SAT_ATI	Saturate (clamp 0..1)

Naturally, `GL_2X_ATI, GL_4X_ATI, GL_8X_ATI, GL_HALF_ATI, GL_QUARTER_ATI` and `GL_EIGHTH_ATI` are mutually exclusive. `GL_SAT_ATI` may be bitwise or'd with any of these or used alone. For no scaling or saturation, use `GL_NONE_ATI`.

Write Masks

A fully general mask may also be applied to writes to the destination register. Any bit combination of `GL_RED_BIT_ATI`, `GL_GREEN_BIT_ATI`, `GL_BLUE_BIT_ATI`, and `GL_ALPHA_BIT_ATI` may be used to mask writes to the destination. Note that, as a destination write mask, `GL_RED_BIT_ATI|GL_GREEN_BIT_ATI|GL_BLUE_BIT_ATI|GL_ALPHA_BIT_ATI` is functionally equivalent to `GL_NONE`.

The sample fragment shader above uses write masks to compute a 2D texture coordinate from the results of two separate calculations. In that case, the red channel of the register is written with the scalar result of N·H and the green channel is set equal to the alpha channel in the bump map. This is result is then as the texture coordinates in a fetch from an $(N{\cdot}H)^k$ map, allowing k to vary across the primitive.

Fragment Shader Output

Unlike the `ATI_vertex_shader` extension, there is no explicit mechanism for specifying the output of the fragment shader. The contents of `GL_REG_0_ATI` are the output of the fragment shader.

Future Directions

Clearly, programmable real-time 3D graphics is the future. We're just beginning to scratch the surface of the visual effects that can be achieved with today's real-time programmable 3D graphics models, but we're already looking at extensions to the functionality outlined here.

For example, in many cases, it would be desirable for a vertex shader to perform some operations on low-polygon pre-tessellated data and perform other operations post-tessellation on the increased number of vertices. Skinning or tweening the control mesh (or input triangulation in the case of Curved PN Triangles) of a game character prior to tessellation might be desirable. Lighting and other "high frequency" calculations would then come after surface tessellation.

Conclusion

We've outlined the behavior of two programmable shading extensions that have been designed to be implementable in hardware in the near term. The APIs are designed to attract multivendor support and to remain in place between multiple generations of hardware implementation. Aside from outlining the behavior of these extensions, we have provided some insight into what motivated many API design decisions.

Acknowledgments

A big thanks to our colleagues at ATI for their valuable input, particularly Dan Ginsburg, Dave Gosselin, Rick Hammerstone, John Isidoro, Steve Morein and Alex Vlachos.

Online References

`ATI_vertex_shader` and `ATI_fragment_shader` specs are online: http://www.ati.com/online/sdk